BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

APPENDIX I

```
/* StorageTek Copyright 1999 */
/*
   load_bal.c simulates an algorithm for balancing the load on data paths
   by changing the assignments of devices to paths.

   Direct questions regarding this program to:
   Richard Defouw
   e_mail: Richard_Defouw@storetek.com
   telephone: 303-673-5817

   OPERATION OF THE PROGRAM
   ========================
   1. Read parameters from the command line.
      This is done by main(). The parameters specified in the command line
      are the following global variables:
      (a) Configuration parameters
          num_paths = number of data paths
          num_dev = number of devices (disks)
      (b) Algorithm parameters (defined with declarations)
          T1
          T2
          move_limit
      (c) Simulation parameters
          num_trials = number of times to execute the balancing algorithm
          (each execution of the algorithm is called a trial or an experiment)
          seed = seed for random number generator
      (d) Mixed (configuration/simulation) parameters (to be explained shortly)
          dev_per_path
          gen_mode
   2. For each trial:
      (a) Create the initial condition (the state that the balancing algorithm
          is to act on). This is done by the function gen_values() in two steps:
          (i) Decide how many devices are allocated to each path in the initial
              state. This is done by first assigning dev_per_path devices to
              each path and then distributing the remaining devices randomly
over
              the paths. Thus, the parameter dev_per_path can be used to control
              how evenly the devices are distributed over the paths. If you want
              the devices distributed as evenly as possible, set dev_per_path =
99.
          (ii) Assign a value to each device. This value is the contribution of
the
               device to the load on the path that is allocated to the device;
that
               is, it is proportional to the amount of data transferred by the
device
               during the most recent time segment. The device values are taken
from
               a random distribution; the distribution is determined by the
parameter
               gen_mode (gen_mode = 2 selects an exponential distribution, which
is
               probably the most realistic of the choices offered).
      (b) Apply the balancing algorithm. This is done by the function balance().
          Pseudocode for the algorithm is provided in load_bal.doc.
      (c) Update the statistics recorded for the trials performed so far.
```

```
   3  Output the statistics summarizing the results of all the trials. The most
important
      statistic shows how effective the path balancing is in reducing the load
on the most
      heavily used path.
*/

#include <stdio.h>
#include <math.h>

#define MAXpaths 100
#define MAXdev 200  /* maximum number of devices on any path */

/* The following constants are used by the random-number generator rand_unif().
*/
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-14
#define RNMX (1.0-EPS)

#define max(A, B)   ((A) > (B) ? (A) : (B))
#define min(A, B)   ((A) < (B) ? (A) : (B))

/* ******************** BEGINNING OF EXTERNAL VARIABLES ******************** */

int num_paths; /* number of paths */
int num_dev;   /* total number of devices */
int dev_per_path; /* minimum number of devices per path at beginning of
experiment:
                     value of 99 means distribute devices as evenly as possible
*/
int move_limit; /* maximum number of devices that may be moved */
int gen_mode; /* 0, 1, 2, or 3; selects method of generating initial device
values */
int num_trials;  /* number of trials (time segments) to simulate */

double Vdev[MAXpaths][MAXdev]; /* Vdev[i][j] is the value of the jth device on
the ith path */
int Ndev[MAXpaths]; /* Ndev[i] is the number of devices assigned to ith path */

double T1; /* load is considered balanced when (hi - lo)/hi <= T1, where hi
              and lo are the largest and smallest path values */
double T2; /* a device is moved from hi path to lo path if new(hi-lo) < T2 *
old(hi-lo) */

/* hi = largest of all path values */
double AggInitHigh; /* aggregate initial hi of all experiments */
double AggFinalHigh; /* aggregate final hi of all experiments */
int Nmoves; /* total number of devices moved in all experiments */
int num_reached_limit; /* number of experiments that reached move_limit */

long seed; /* for random-number generator rand_unif() */
```

```
/* ********************* END OF EXTERNAL VARIABLES ********************** */


/* ************************ BEGINNING OF main() ************************** */

main(int argc, char *argv[])
{
int i;

num_paths = atoi(argv[1]);
num_dev = atoi(argv[2]);
dev_per_path = atoi(argv[3]); /* 99 means distribute devices evenly */
T1 = atof(argv[4]);  /* enter as a percentage */
T2 = atof(argv[5]);  /* enter as a percentage */
move_limit = atoi(argv[6]);
gen_mode = atoi(argv[7]);   /* 0, 1, 2, or 3 */
num_trials = atoi(argv[8]);
seed = -atoi(argv[9]);

if (num_paths > MAXpaths) {
    printf("TOO MANY PATHS\n");
    exit(0);
}
if (dev_per_path != 99) {
    if (num_dev < num_paths * dev_per_path) {
        printf("NOT ENOUGH DEVICES\n");
        exit(0);
    }
}

printf("\nBALANCING %d DEVICES ON %d PATHS\n",num_dev,num_paths);
printf("Threshold for judging balance = %.1lf%%\n",T1);
printf("Threshold for judging whether a move is worthwhile = %.1lf%%\n",T2);
printf("Maximum number of devices that can be moved = %d\n",move_limit);
T1 /= 100; /* convert from percentage to fraction */
T2 /= 100; /* convert from percentage to fraction */
if (dev_per_path == 99) {
    printf("Devices distributed as evenly as possible.\n");
}
else {
    printf("At least %d devices per path\n",dev_per_path);
}
if (gen_mode == 0) {
    printf("Device values are CONSTANT.\n");
}
else if (gen_mode == 1) {
    printf("Device values are UNIFORMLY distributed.\n");
}
else if (gen_mode == 2) {
    printf("Device values are EXPONENTIALLY distributed.\n");
}
else if (gen_mode == 3) {
    printf("Device values are based on POWER-LAW function of rank. \n");
}
else {
    printf("INVALID gen_mode\n");
```

```
    exit(0);
}
printf("Number of trials = %d; seed = %ld\n",num_trials,-seed);

Nmoves = 0;
num_reached_limit = 0;
AggFinalHigh = 0;
AggInitHigh = 0;

for (i = 0; i < num_trials; i--) {
    /* generate device values and assignment of devices to paths */
    gen_values();

    /* balance load */
    balance();
}

/* print results */
printf("Mean final high/mean initial high =
%.2lf%%\n",100*AggFinalHigh/AggInitHigh);
printf("Mean number of devices moved = %.3lf\n",(double)Nmoves/num_trials);
printf("Number of experiments that reached move limit =
%d\n",num_reached_limit);

}
/* ***************************** END OF main() ************************** */


/* ********************* BEGINNING OF gen_values() ************************
*/

/* This function sets up the conditions for each experiment; that is, it
   generates device values for a given time segment. */

gen_values()
{
int extra, check; /* variables used in assignment of devices to paths */
int separator[MAXpaths]; /* array used in assignment of devices to paths */
int num_separators; /* number of separator[] elements that have been computed */
int sorted_sep[MAXpaths]; /* sorted version of separator[] */
int smallest, last; /* variables used in sorting separator[] */
int i,j,k;
/* following 4 variables are used to generate random permutation of skew ranks
when gen_mode = 3 */
int permute_source[1000];
int permute_size; /* number of unused elements in permute_source[] */
int index;
double rank;

double rand_exp(); /* used for gen_mode = 2 */
double rand_unif();

for (i = 0; i < MAXpaths; i++) {
    for (j = 0; j < MAXdev; j++) {
        Vdev[i][j] = 0;
    }
}
```

```
for (i = 0; i < MAXpaths; i++) {
    Ndev[i] = 0;
}

/* assign devices to paths */
if (dev_per_path == 99) {
    /* devices are to be distributed as uniformly as
       possible over the paths */
    dev_per_path = num_dev/num_paths;
    extra = num_dev - num_paths * dev_per_path;
    for (i = 0; i < extra; i++) {
        Ndev[i] = dev_per_path + 1;
    }
    for (i = extra; i < num_paths; i++) {
        Ndev[i] = dev_per_path;
    }
}
else {
    for (i = 0; i < num_paths; i++) {
        Ndev[i] = dev_per_path;
    }
    extra = num_dev - num_paths * dev_per_path;
    /* randomly assign the extra devices to paths */
    extra += num_paths - 1;
    for (num_separators = 0; num_separators < num_paths - 1; ++num_separators) {
        separator[num_separators] = rand_unif() * extra;
        for (i = 0; i < num_separators; i++) {
            if (separator[i] == separator[num_separators]) {
                --num_separators;
                break;
            }
        }
    }
    /* sort separator[] */
    last = -1;
    for (i = 0; i < num_paths - 1; i++) {
        smallest = extra;
        for (j = 0; j < num_paths - 1; j++) {
            if (separator[j] < smallest && separator[j] > last) {
                smallest = separator[j];
            }
        }
        sorted_sep[i] = smallest;
        last = smallest;
    }
    /* use the separators to distribute the extra devices to the paths */
    Ndev[0] += sorted_sep[0];
    for (i = 1; i < num_paths - 1; i++) {
        Ndev[i] += sorted_sep[i] - 1 - sorted_sep[i-1];
    }
    Ndev[num_paths - 1] += extra - 1 - sorted_sep[num_paths - 2];
}

/* check assignment of devices to paths */
check = 0;
for (i = 0; i < num_paths; i++) {
    if (Ndev[i] > MAXdev) {
```

```
            printf("TOO MANY DEVICES ON ONE PATH\n");
            exit(0);
        }
        check += Ndev[i];
    }
    if (check != num_dev) {
        printf("ERROR IN ASSIGNMENT OF DEVICES TO PATHS\n");
        exit(0);
    }

    /* compute the device values */
    if (gen_mode == 1) {
        for (i = 0; i < num_paths; i++) {
            for (j = 0; j < Ndev[i]; j++) {
                Vdev[i][j] = rand_unif();
            }
        }
    }
    else if (gen_mode == 2) {
        for (i = 0; i < num_paths; i++) {
            for (j = 0; j < Ndev[i]; j++) {
                Vdev[i][j] = rand_exp();
            }
        }
    }
    else if (gen_mode == 3) {
    /* Use formula for skew as function of rank given by Peterson and Grossman in
       "Power Laws in Large Shop DASD I/O Activity", CMG Proceedings for 1995. p.p.
       822-833. Figure 1; assign ranks to devices by generating a random
    permutation. */
        for (i = 0; i < num_dev; i++) {
            permute_source[i] = i + 1;
        }
        permute_size = num_dev;
        for (i = 0; i < num_paths; i++) {
            for (j = 0; j < Ndev[i]; j++) {
                index = rand_unif() * permute_size;
                rank = permute_source[index];
                Vdev[i][j] = pow(rank + 1.94, -1.26);
                --permute_size;
                for (k = index; k < permute_size; k++) {
                    permute_source[k] = permute_source[k+1];
                }
            }
        }
    }
    else if (gen_mode == 0) {
    /*
       This option was implemented for testing purposes. In particular, this
       program run with a sufficiently large number of trials for num_paths = 2,
       num_dev even, dev_per_path = 0, and gen_mode = 0 yields results that agree
       with the following expected solution:
           (mean final hi)/ (mean initial hi) = 2(N + 1)/(3N + 4)
           mean number of moves = 0.25N(N + 2)/(N + 1),
       where N stands for num_dev.
    */
        for (i = 0; i < num_paths; i++) {
```

```
        for (j = 0; j < Ndev[i]; j++) {
            Vdev[i][j] = 0.1;
        }
    }
}
}

/* ************************** END OF gen_values() ****************************
*/


/* ********************** BEGINNING OF balance() ********************** */

balance()
{
double Vpath[MAXpaths]; /* Vpath[i] is the total value of the devices on the ith
path */
double hi;  /* largest of all Vpath[i] */
double lo;  /* smallest of all Vpath[i] */
int num_moved; /* number of devices moved in current experiment */
int hi_path; /* the path with Vpath[i] = hi */
int lo_path; /* the path with Vpath[i] = lo */
double trgt; /* trgt is the "optimum" device value to be moved from hi path to
lo path;
                algorithm uses trgt = (hi-lo)/2 */
int trgt_dev; /* the next device to be moved has value Vdev[hi_path][trgt_dev]
*/
double x, delta; /* variables used in finding trgt_dev */
int i,j;

for (i = 0; i < num_paths; i++) {
    Vpath[i] = 0;
    for (j = 0; j < Ndev[i]; j++) {
        Vpath[i] += Vdev[i][j];
    }
}

lo = 9999999999999999;
for (i = 0; i < num_paths; i++) {
    lo = min(lo, Vpath[i]);
}

hi = 0;
for (i = 0; i < num_paths; i++) {
    hi = max(hi, Vpath[i]);
}

AggInitHigh += hi;

num_moved = 0;

while (hi - lo > TL * hi && num_moved < move_limit) {
    trgt = (hi - lo) / 2;

    /* identify the hi and lo paths */
    for (i = 0; i < num_paths; i++) {
        if (Vpath[i] == hi) {
```

```
            hi_path = i;
            break;
        }
    }
    if (i >= num_paths) {
        printf("ERROR IN FINDING EXTREME PATH\n");
        exit(1);
    }
    for (i = 0; i < num_paths; i++) {
        if (Vpath[i] == lo) {
            lo_path = i;
            break;
        }
    }
    if (i >= num_paths) {
        printf("ERROR IN FINDING EXTREME PATH\n");
        exit(1);
    }

    /* identify the next device to be moved, if there is one */
    delta = 9999999999999999;
    trgt_dev = Ndev[hi_path];
    for (i = 0; i < Ndev[hi_path]; i++) {
        x = Vdev[hi_path][i];
        if (fabs(x - trgt) < T2 * trgt && fabs(x - trgt) < delta) {
            trgt_dev = i;
            delta = fabs(x - trgt);
        }
    }
    if (trgt_dev == Ndev[hi_path]) {
    /* hi path contains no device that can profitably be moved; terminate
balancing */
        break;
    }

    /* we have found the device we wish to move */
    /* add this device to lo path */
    Vdev[lo_path][Ndev[lo_path]] = Vdev[hi_path][trgt_dev];
    Vpath[lo_path] += Vdev[hi_path][trgt_dev];
    ++Ndev[lo_path];

    /* remove device from hi path */
    Vpath[hi_path] -= Vdev[hi_path][trgt_dev];
    --Ndev[hi_path];
    for (i = trgt_dev; i < Ndev[hi_path]; i++) {
        Vdev[hi_path][i] = Vdev[hi_path][i+1];
    }
    Vdev[hi_path][Ndev[hi_path]] = 0;

    /* compute new values of hi and lo */
    lo = 9999999999999999;
    for (i = 0; i < num_paths; i++) {
        lo = min(lo, Vpath[i]);
    }

    hi = 0;
    for (i = 0; i < num_paths; i++) {
```

```
            hi = max(hi, Vpath[i]);
        }

        ++num_moved;
    }

    if (num_moved >= move_limit) {
        ++num_reached_limit;
    }
    Nmoves += num_moved;
    AggFinalHigh += hi;
    }

    /* ************************** END OF balance() ************************** */


    /* ********************* BEGINNING OF rand_exp() ************************* */

    /* This function is taken from page 287 of "Numerical Recipes in C"
       (second edition) by Press, Teukolsky, Vetterling, and Flannery; it
       returns an exponential deviate with mean 1.0 calculated from a uniform
       deviate generated by rand_unif().*/

    double rand_exp()
    {
    double rand_unif();
    double dum;

    do {
        dum = rand_unif();
    } while (dum == 0.0);

    return -log(dum);
    }

    /* ************************** END OF rand_exp() ************************** */


    /* ********************* BEGINNING OF rand_unif() ************************ */

    /* This function is taken from page 280 of "Numerical Recipes in C"
       (second edition) by Press, Teukolsky, Vetterling, and Flannery; it
       returns a uniform deviate in (0,1). */

    double rand_unif()
    {
    int j;
    long k;
    static long iy = 0L;
    static long iv[NTAB];
    double temp;

    if (seed <= 0L || !iy) {            /* Initialize */
        if (-seed < 1L) {               /* Be sure to prevent seed = 0 */
            seed = 1L;
        }
        else {
```

```
        seed = -seed;
    }
    for (j = NTAB+7; j >= 0; j--) { /* Load the shuffle table after 8 warmups */
        k = seed/IQ;
        seed = IA*(seed - k*IQ) - IR*k;
        if (seed < 0L) {
            seed += IM;
        }
        if (j < NTAB) {
            iv[j] = seed;
        }
    }
    iy = iv[0];
}

k = seed/IQ;                        /* Start here when not initializing */
seed = IA*(seed - k*IQ) - IR*k;
if (seed < 0L) {
    seed += IM;
}
j = iy/NDIV;
iy = iv[j];
iv[j] = seed;
if ((temp = AM*iy) > RNMX) {
    return RNMX;                    /* Because users don't expect endpoint values */
}
else {
    return temp;
}
}

/* ************************** END OF rand_unif() ************************** */
```